Attorney's Docket No.: 42390.P13132 *PATENT*

APPLICATION FOR UNITED STATES LETTERS PATENT

For

# SYSTEM AND METHOD TO REDUCE THE SIZE OF EXECUTABLE CODE IN A PROCESSING SYSTEM

Inventor:

**ARCH D. ROBISON**

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Seventh Floor
Los Angeles, CA 90025-1026
(408) 947-8200

"Express Mail" mailing label number: EV031348745US
Date of Deposit: January 3, 2002
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Commissioner for Patents, Washington, DC 20231
Patricia M. Richard
(Typed or printed name of person mailing paper or fee)
PRichard
(Signature of person mailing paper or fee)
January 3, 2002
(Date signed)

# SYSTEM AND METHOD TO REDUCE THE SIZE OF EXECUTABLE CODE IN A PROCESSING SYSTEM

## FIELD OF THE INVENTION

[0001] The present invention relates generally to compiler systems and, more particularly, to a system and method to reduce the size of executable code in a processing system.

## BACKGROUND OF THE INVENTION

[0002] Increasingly, the size of compiled code has an impact on the performance and economics of computer systems. From embedded systems, such as cellular phones, to applets shipped over the World Wide Web, the impact of compile-time decisions that expand the size of the executable code has a direct effect on cost and power consumption, as well as on the transmission and execution time of the code.

[0003] Several techniques have been proposed to reduce the size of the executable code. One known technique turns repeated code fragments into procedures and is usually applied to intermediary code or even source code. However, this technique appears to miss repeated code fragments introduced during code generation. Another known technique reuses the common tail of two merging code sequences and is usually performed after code generation. However, syntactic mismatches seem to affect the efficiency of this technique.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0004] The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

[0005] **Figure 1** is an exemplary executable code within a program.

[0006] **Figure 2A** is a block diagram of one embodiment of a graph structure representing the executable code shown in **Figure 1**.

[0007] **Figure 2B** is a block diagram of one embodiment of the graph structure showing a reduced executable code obtained through a method to reduce the size of the executable code in a processing system.

[0008] **Figure 3** is a flow diagram of one embodiment of the method to reduce the size of the executable code.

[0009] **Figure 4A** is a block diagram of one embodiment of an interference graph structure constructed in connection with the executable code.

[0010] **Figure 4B** is a table illustrating the interference graph structure shown in **Figure 4A**.

[0011] **Figure 5** is a block diagram of a data dependence graph structure constructed in connection with the executable code.

[0012] **Figure 6** is a flow diagram of one embodiment of a method to transfer unifiable instructions from the tines to the handle of the graph structure representing the executable code.

[0013] **Figure 7** is a block diagram of one embodiment of the processing system.

## DETAILED DESCRIPTION

[0014] In the following descriptions for the purposes of explanation, numerous details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that these specific details are not required in order to practice the present invention. In other instances, well-known electrical structures or circuits are shown in block diagram form in order not to obscure the present invention unnecessarily.

[0015] A system and method to reduce the size of executable code in a processing system are described. Multiple subgraph structures are identified within a graph structure constructed for multiple executable instructions in a program. Unifiable variables that are not simultaneously used in the executable instructions are identified within each subgraph structure. Finally, one or more unifiable instructions from a tine of a corresponding subgraph structure are transferred to a handle of the corresponding subgraph structure, each unifiable instruction containing one or more unifiable variables.

[0016] **Figure 1** is an exemplary executable code within a program. As illustrated in **Figure 1**, in one embodiment, the exemplary executable code 100 includes multiple lines, each line containing a separate executable instruction. The executable instructions can be at a low backend level, or at some higher level closer to the front end. As shown in **Figure 1**, the executable code 100 includes an "if(p)" clause and an "else" clause, each containing multiple executable instructions.

[0017] **Figure 2A** is a block diagram of one embodiment of a graph structure representing the executable code shown in **Figure 1**. As illustrated in **Figure 2A**, in one embodiment, graph structure 200 includes a common predecessor "if(p)" block 201, a subgraph structure containing two tines 202, 203, for example sequences of Instructions with straight-line control flow, also known as basic blocks, and a common successor "return z" block 204. The basic block 202 contains instructions located in the "if(p)" clause of the executable code 100 and the basic block 203 contains instructions located in the "else" clause. The two basic blocks 202, 203 share two handles, the common predecessor block 201 and the common successor block 204.

[0018] In one embodiment, if a known "omega motion" procedure is used, wherein executable instructions are moved downwardly past join points, matching instructions within the basic blocks 202, 203 are unified and subsequently transferred to common successor block 204, while being removed from their respective locations in the basic blocks 202 and 203. Alternatively, if a known "alpha motion" procedure is used, wherein executable instructions are moved upwardly past fork points, matching instructions within the basic blocks 202, 203 are unified and subsequently transferred to common predecessor block 201, while being removed from their respective locations within the blocks 202 and 203.

[0019] For example, in the case of an "omega motion" procedure, the instruction "z = x*y", located within basic block 202, and the instruction "z = a*b", located within basic block 203, may be unified by unifying the variables x

and b, and unifying the variables y and a, and by exploiting the fact that each multiplication is commutative. Once the two unifiable instructions are unified, the pair of matching instructions may be removed from the basic blocks 202, 203 and a single copy of the unified instruction may be inserted past the join point. The process of identification and unification of the matching unifiable instructions will be described in further detail below.

[0020] **Figure 2B** is a block diagram of one embodiment of the graph structure showing a reduced executable code obtained through a method to reduce the size of the executable code in a processing system. As illustrated in **Figure 2B**, the instruction "z = x*y", located within basic block 202, and the instruction "z = a*b", located within basic block 203, are unified by unifying the variables x and b, and unifying the variables y and a. A unified instruction "z = x*y" is transferred to the common successor block 204. The process of transferring the unified instruction to a common handle, for example common successor block 204, will be described in further detail below.

[0021] **Figure 3** is a flow diagram of one embodiment of the method to reduce the size of the executable code. As illustrated in **Figure 3**, at processing block 310, fork subgraph structures are identified within a graph structure, which represents the executable code. In one embodiment, the executable code 100 is scanned for candidate fork subgraph structures. Each fork subgraph structure contains multiple tines or basic blocks, which share a common successor handle or block (in the case of the omega motion procedure) or which

share a common predecessor handle or block (in the case of the alpha motion procedure).

[0022] At processing block 320, an interference graph structure is constructed for local variables mentioned on the tines of each subgraph structures. In one embodiment, the interference graph structure indicates which variables of the local variables are simultaneously used in the executable instructions within the tines and cannot be unified, i.e., the variables that have overlapping live ranges.

[0023] The interference graph structure is represented as a symmetric matrix INTERFERE (i,j), where i and j correspond to local variables. INTERFERE(i,j,) is true if variables i and j are both live at some point in the program. In one embodiment, determining whether the variables have overlapping live ranges can be achieved using one of many known methods of determination. One of such methods is data-flow analysis. In the exemplary executable code of **Figure 1**, data-flow analysis can determine that a and y are local to different blocks 201 through 204. Another example of such methods is syntactic analysis. Syntactic analysis can determine that b and x are local to their respective blocks, because they are locally scoped.

[0024] **Figure 4A** is a block diagram of one embodiment of an interference graph structure constructed in connection with the executable code. As illustrated in **Figure 4A**, in one embodiment of the interference graph structure 400, node 401 represents local variable a, node 402 represents local variable b, node 403 represents local variable x, node 404 represents local variable y, and node 405 represents local variable z. Variables a and b interfere, respective

nodes 401 and 402 being connected to each other. Similarly, variables x and y interfere, respective nodes 403 and 404 being connected to each other.

**[0025]** Variable z does not interfere with any other local variable. For example, variable z is assigned immediately after x and y are last used, so the live range of variable z abuts, but does not overlap, the live ranges of variables x and y. Global variables, for example variable w, are not included in the interference graph structure 400 because they are presumed to have unknown live ranges. In one embodiment, the interference graph structure 400 also includes self loops for each local variable x, y, z, a, and b.

**[0026]** **Figure 4B** is a table illustrating the interference graph structure shown in **Figure 4A**. As illustrated in **Figure 4B**, a "1" value is assigned to any table entry, which has its row and column corresponding to local variables having overlapping live ranges, for example a and b, x and y. A "1" value is assigned to each entry on the main diagonal of the table to account for the self loops.

**[0027]** Referring back to **Figure 3**, at processing block 330, a data dependence graph structure is constructed for the executable code. In one embodiment, for an alpha motion procedure, the data dependence graph structure has a directed arc u v, if the instruction u must precede the instruction v, typically because the instruction v uses a value computed by the instruction u, i.e. instruction v depends upon instruction u. Alternatively, for an omega motion procedure, the data dependence graph structure has a directed arc u v, if the instruction u must succeed the instruction v. An arc u v is said to be "properly oriented" within a tine t if instructions u and v belong to tine t, and if, when performing alpha

motion, instruction u precedes v, or when performing omega motion, instruction u succeeds instruction v. (An improperly oriented arc can arise from loop-carried dependencies.)

[0028] **Figure 5** is a block diagram of a data dependence graph structure constructed in connection with the executable code. As illustrated in Figure 5, the data dependence graph structure 500 shows the directed edges between any two instructions within the exemplary executable code 100. For example, considering an alpha motion procedure, the foo(&x) instruction must precede the z=x*y instruction, the y=3 instruction must precede the z=x*y instruction, the a=3 instruction must precede the z=a*b instruction, and the foo(&b) instruction must precede the z=a*b instruction.

[0029] Referring back to **Figure 3**, at processing block 340, for each subgraph structure, unifiable instructions are transferred from corresponding tines to a handle, as described in further detail below.

[0030] **Figure 6** is a flow diagram of one embodiment of a method to transfer unifiable instructions from the tines to the handle of the graph structure representing the executable code. As illustrated in **Figure 6**, at processing block 610, for each tine of a subgraph structure, for each instruction in the tine, dependence arcs are counted and a flag is initialized for each instruction that may be transferred. One embodiment of processing block 610 is implemented in pseudo-code as follows. If all elements of COUNTER are assumed to be previously initialized to zero:

[0031] For each tine t of fork
    For each instruction x on tine t

```
        For each dependence arc with tail x do
            If arc is properly oriented within tine t then
                Let y be the head of the arc
                COUNTER[y] := COUNTER[y]+1;
    For each instruction x on tine t
        If COUNTER[x]==0 then
            READY[t] = READY[t] ∪ {x}
```

[0032] At processing block 620, a decision is made whether a search for the flags returns true, i.e. initialized flags are found. One embodiment of processing block 620 is implemented in pseudo-code as follows:

[0033]
```
procedure FIND_READY() {
    Let s be the first tine.
    r := depth of transaction stack
    For each instruction x in READY[s] do {
        CANDIDATE[s] = x
        For each tine t in a tine distinct from tine s do {
            For each instruction y in READY[t] do {
                if ARE_INSTRUCTIONS_UNIFIABLE (x,y) then {
                    CANDIDATE[t] = y
                    goto next_t
                }
            }
            goto next_x
            next_t:
        }
        // We have a set of unifiable instructions.
        return true;

        next_x:
            roll back transaction stack to depth r
    }
```

[0034]
```
boolean ARE_INSTRUCTIONS_UNIFIABLE(x,y) {
    if x and y have the same operator then {
        r := depth of transaction stack
            for each corresponding operand k of x and y do {
                if not ARE_OPERANDS_UNIFIABLE( x[k], y[k] ) {
                    roll back transaction stack to depth r
                    return false
                }
```

```
            }
            return true
        } else {
            return false;
        }
    }
```

[0035]
```
boolean ARE_OPERANDS_UNIFIABLE(a,b) {
    If a and b are not the same kind of operand, return false;
    If a and b do not have equivalent types, return false;
    switch kind of a:
        LITERAL:
          if b has same value as a return true
          else return false;
        INDIRECTION:
          if base pointers for a and b are unifiable return true
          else return false;
        VARIABLE:
          Let A be union-find representative for a.
          Let B be union-find representative for b.
          It A==B then return true;
          If A and B are local variables and not
CHECK_FOR_INTERFERENCE(a,b) then {
            Tentatively unify A and B
            Push transaction for the tentative unification onto the
transactions stack.
                return true;
          } else {
            return false;
          }
    end switch
    return false;
}
```

[0036]
```
boolean CHECK_FOR_INTERFERENCE(a,b) {
    for each element i in union-find set of a do
        for each element j in union-find set of b do
            if INTERFERE(i,j)
                return false
    return true;
}
```

[0037] A "transaction stack" is a stack of information about tentative unifications that might have to be rolled back, i.e. undone. In the pseudo-code,

the phrase "roll back transaction stack to depth r" means to iteratively pop transactions, and undo their effects, until the stack has depth r again.

**[0038]** In order to match variables, a look up for their union-find representatives is performed to assess if the representatives are equal. If so, the variables are already identical, or have already been unified (possibly tentatively). Otherwise, a check is performed to assess if the variables' union-find representatives do not interfere (i.e. have disjoint live ranges). If two variables do not interfere, then the variables are tentatively unified. A transaction for this unification is then pushed onto the transaction stack, so that it can be undone at a later time if considered necessary.

**[0039]** If no flags are found and the search returns false, then the procedure ends. Otherwise, if the search for flags returns true, at processing block 630, the instructions associated with the initialized flags are unified. One embodiment of processing block 630 is implemented in pseudo-code as follows:

**[0040]**
```
procedure DO_REQUIRED_UNIFICATIONS {
    for each transaction stack entry u from bottom to top
        do unification specified by u
    set transaction stack to empty
}
```

**[0041]** At processing block 640, for each tine, dependence counts for the instructions dependent on the unified instructions are decremented. One embodiment of processing block 640 is implemented in pseudo-code as follows:

**[0042]**
```
For each tine t of fork
    For each dependence arc with tail CANDIDATE[t] do
        If arc is properly oriented within tine t then {
            Let y be the head of the arc
            COUNTER[y] = COUNTER[y]-1;
            if COUNTER[y]==0 then
```

READY[t] = READY[t] ∪ {y}

}

[0043] At processing block 650, the unified instruction is moved from the first corresponding tine to the handle. Finally, at processing block 660, any unified instruction is removed from any subsequent corresponding tines. Processing blocks 620 through 660 are subsequently repeated.

[0044] **Figure 7** is a block diagram of one embodiment of the processing system. As illustrated in Figure 7, processing system 700 includes a memory 710 and a processor 720 coupled to the memory 710. In some embodiments, the processor 720 is a processor capable of compiling software and annotating code regions of the program. Processor 720 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. The processing system 700 can be a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other system that includes software.

[0045] Memory 710 can be a hard disk, a floppy disk, random access memory (RAM), read only memory (ROM), flash memory, or any other type of machine medium readable by the processor 720. Memory 710 can store instructions to perform the execution of the various method embodiments of the present invention.

[0046] In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the

appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense.